UNIT 2: Methods for Routine Design

- Kinds of Design Tasks
- Example of Routine Design: Gear Boxes
- Simple Problem Spaces Don't Work
- Mixing Constraint Propagation with Search
- Implementation Using Rule-Based Programming
- Other Forms of Rule-Based Programming

Design Tasks

Given: Functional requirements

Spatial and appearance constraints

Material and manufacturing constraints

Cost, manufacturability, and other optimization criteria

Find: A device that satisfies the functional requirements and the given constraints.

Kinds of Design Tasks

Design tasks can be classified along many dimensions:

- **Amount of knowledge of the design space**
  - **Creative:** little or no prior knowledge.
  - **Nonroutine:** moderate prior knowledge.
  - **Routine:** large amount of prior knowledge.
- **Complexity of the search space**
  - **Catalog Selection:** pre-enumerated complete solutions.
  - **Configuration:** Select and arrange pre-enumerated components. Combinatorial.
  - **New components or systems:** Must consider all possible shapes of basic materials. Continuous and unbounded.
- **Size of space of acceptable solutions**

Kinds of Design Tasks (2)

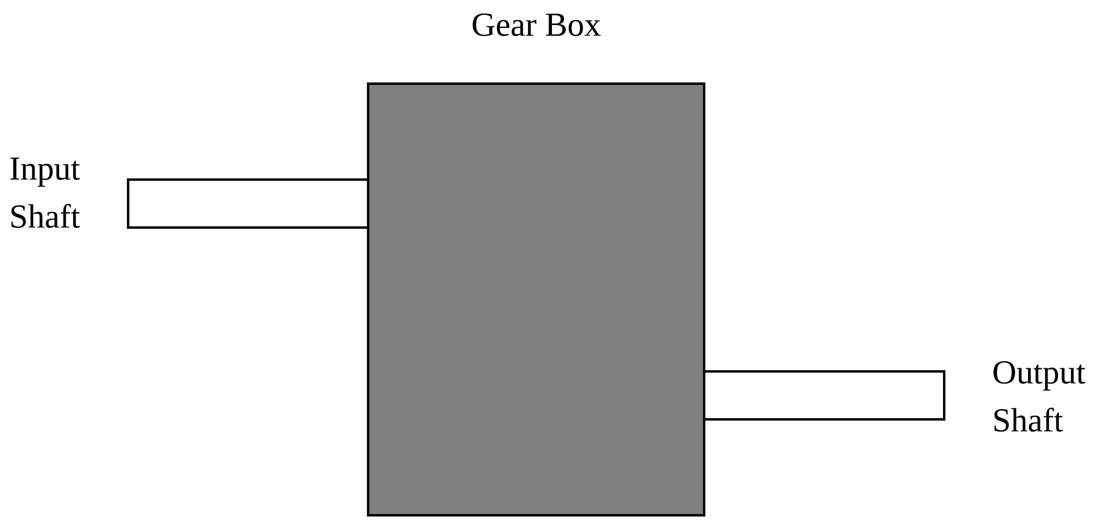
- **Nature of constraint interactions**
  - **Local constraints:** Can be solved by local propagation.
  - **Global constraints:** Can't be applied until design is nearly complete.
- **Phase of design process**.
  - **Conceptual:** Earliest phase where basic approach is chosen.
  - **Layout:** Identification of subsystems and subcomponents and their relationships.
  - **Detailed:** Detailed design of individual subsystems and interfaces.

We will consider routine configuration design. This involves a mix of local and global constraints, and it usually occurs during the detailed design phase.

Examples of Routine Configuration Design

- **A la carte manufacturing.** Computer vendors offer a wide range of options for CPU, memory, disk, bus slots, monitors, multimedia devices, etc. There can be hundreds of thousands of possible configurations. This raises two problems: constructing a design to meet customer needs and checking that the design will actually work. DEC built XCON, a configuration design tool for the DEC Vax computer line. They also built XSEL, a tool for helping sales people price out possible configuration alternatives.
- **Design of digital circuits.** Given a library of standard component cells and various power and space requirements, design a digital circuit to achieve a desired function.
- **Design of blends.** In the coffee and petroleum industries, the retail product is a blend of several raw materials. Many alternative blends may achieve the same desired level of quality. The goal is to choose a blend to minimize cost while maintaining quality.

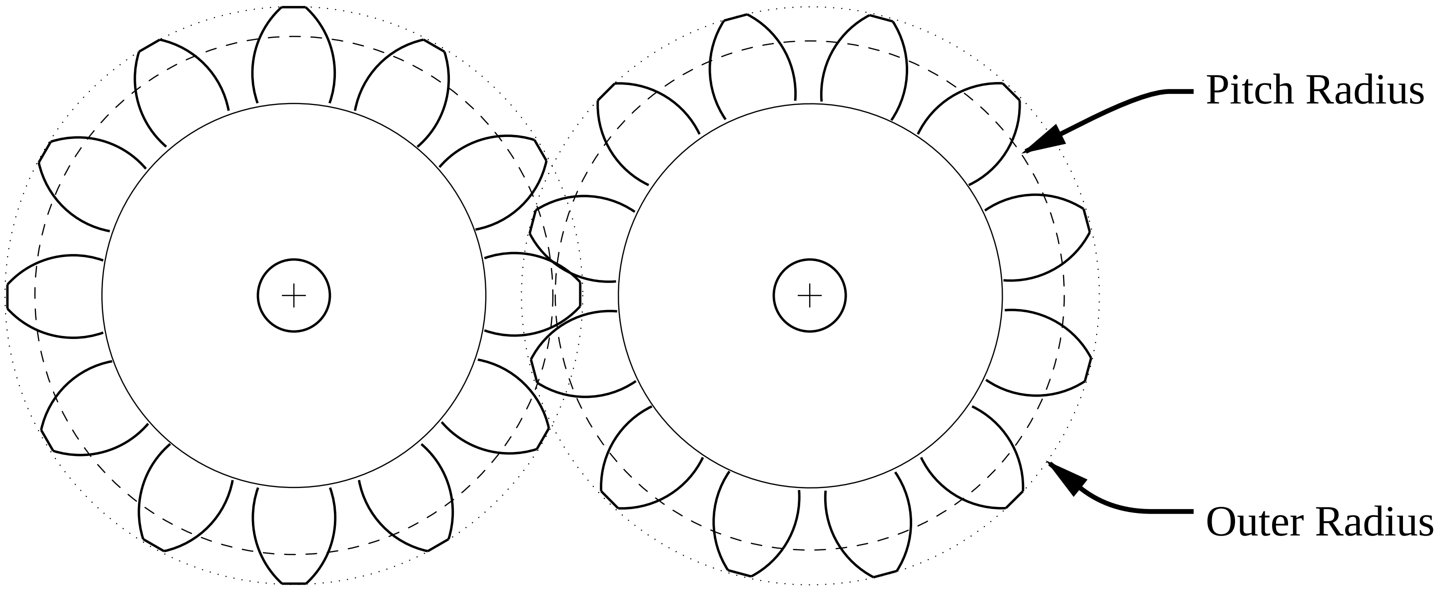
Example Routine Configuration Design Task: Gear Box Design



Given: Desired ratio between input and output revolutions.
Dimensions of gear box enclosure.
Desired positions of the shafts (within some tolerances).
Catalog of possible gears and shafts.

Find: Locations of any idler shafts.
Type and position of each gear on each shaft.

Gears



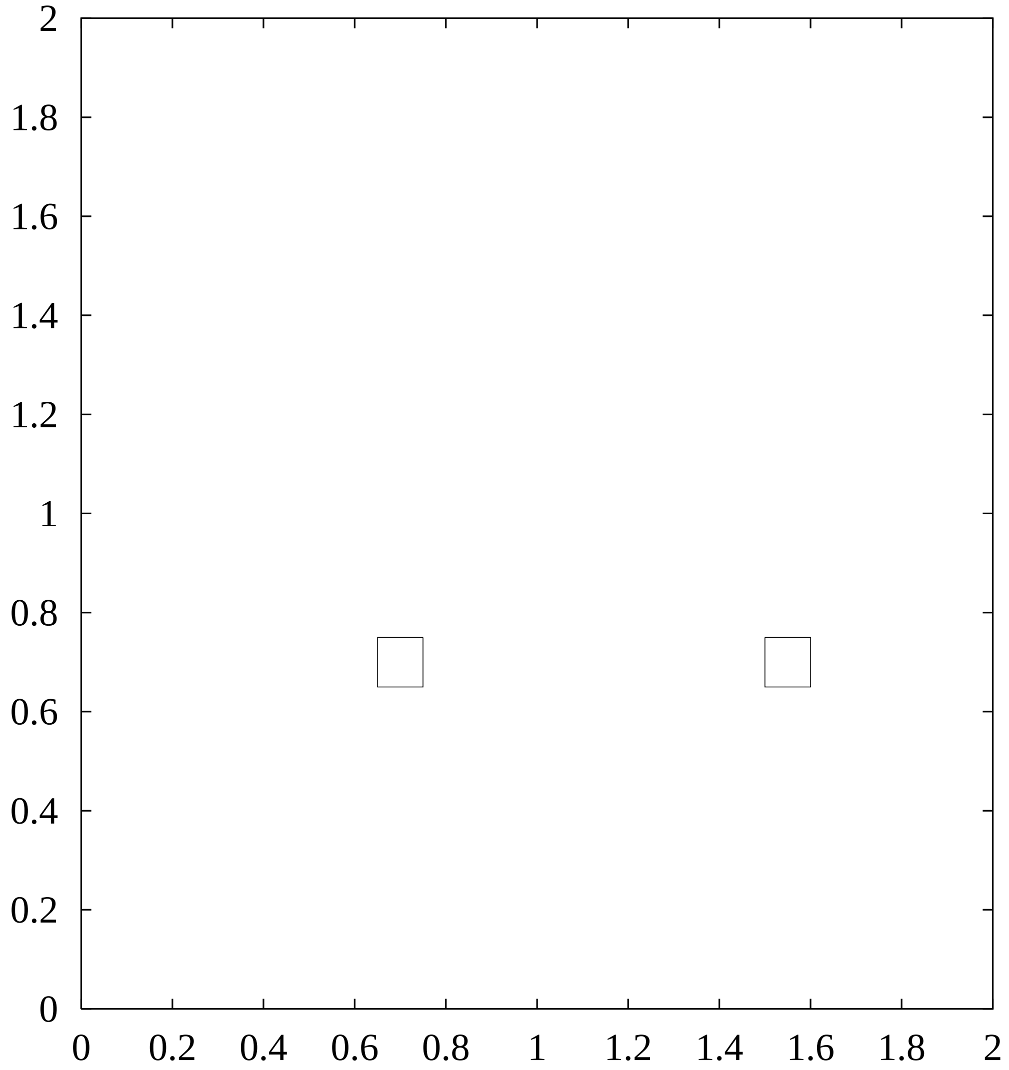
The effective distance between two gears with *pitch radii* R_1 and R_2 is $R_1 + R_2$.

However, a gear will collide with another part if that part is closer than the *outer radius*.

If gear 1 has N teeth and gear 2 has M teeth, then the ratio of revolutions between them is N/M.

An Example Design Problem

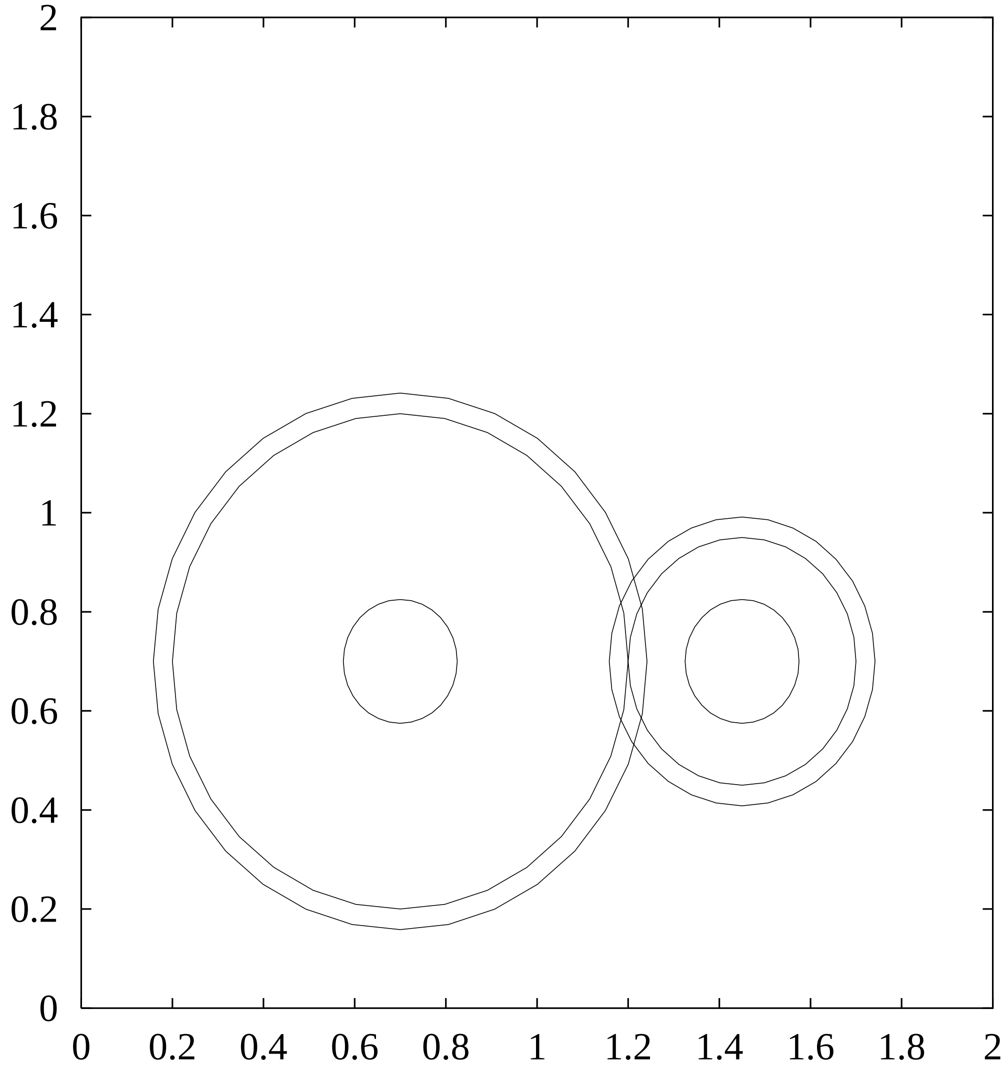
Suppose we have the following design problem. The goal is to achieve a 2/1 ratio between inputs and outputs. The small boxes indicate the allowed positions of the gears.



Available Gears

# Teeth	Pitch Diameter	Outer Diameter	# Teeth	Pitch Diameter	Outer Diameter
12	0.5000	0.583	48	2.0000	2.083
15	0.6250	0.708	54	2.2500	2.333
16	0.6667	0.750	60	2.5000	2.583
18	0.7500	0.833	66	2.7500	2.833
20	0.8333	0.917	69	2.8750	2.958
21	0.8750	0.958	72	3.0000	3.083
22	0.9167	1.000	75	3.1250	3.208
24	1.0000	1.083	78	3.2500	3.333
27	1.1250	1.208	81	3.3750	3.458
30	1.2500	1.333	84	3.3500	3.583
33	1.3750	1.458	87	3.6250	3.708
36	1.5000	1.583	90	3.7500	3.833
39	1.6250	1.708	93	3.8750	3.958
42	1.7500	1.833	96	4.0000	4.083
45	1.8750	1.958	99	4.1250	4.208

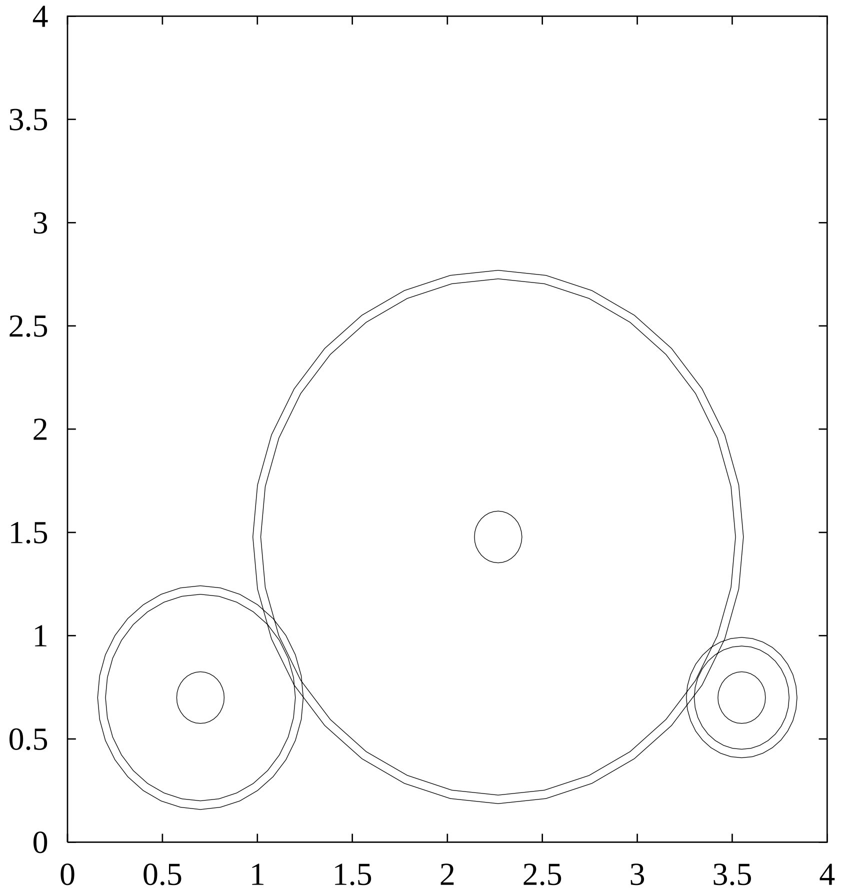
A Solution



The input gear has 24 teeth and the output gear has 12 teeth.

Another Example

The solution to this problem involves introducing an “idler” shaft with one gear on it.



Problem Space Search

- **State:** a design in which some parts have been selected and given locations. Each shaft has two possible positions (A and B) at which a gear can be mounted. A shaft can be located anywhere.
- **Feasible Solution:** A state containing a chain of meshing gears or shared shafts from input to output shaft giving the desired ratio. Input and output shafts are positioned within given regions. Gears do not collide with other gears, shafts, or the box.
- **Start State:** A state containing only the box.
- **Operators:** There are two operators:
  - Add Shaft(location) adds a new shaft at the indicated location.
  - Add gear(shaft, position) adds a new gear on the indicated shaft at the indicated position.
- **Objective Function:** Minimize number of distinct components and total number of components.

Analysis of this Problem Space

- **Branching Factor**. There are finitely many gears, but there are infinitely many possible locations for each shaft. We must discretize the space in order to compute with it.
- **Solution Depth**. The simplest problems have solutions at depth 4. A problem with 3 idler shafts each having 2 gears would have a solution at depth 11.
- **Solution Density**. Extremely low (even in a discretized space).
- **Redundancy.** The space is redundant. Shafts can be constructed in any order, gears can be placed on them in any order.
- **Completeness.** The space is complete.

Why is this space bad?

The operators do not incorporate constraints on feasible solutions.

- Gears along chain must achieve desired ratio. [Somewhat global. Independent of shaft locations or positions of gears on shafts.]
- Gears must not collide with box. [Nearly local. For very large gears, can be evaluated immediately. For most gears, can't be evaluated until gear is mounted on shaft.]
- Gears must not collide with other gears. [Intermediate. Cannot be fully evaluated for one gear until all other gears have been chosen and positioned on shafts that have been located. However, whenever two gears have been placed on located shafts, the constraint can be partially tested.]
- Gears must not collide with other shafts. [Intermediate. Partially evaluable when a new shaft is positioned.]
- Each gear must mesh with (or share a shaft with) the next gear in the chain. [local]
- Chain must connect input shaft with output shaft. [global]

The Evil of Generate-and-Test Behavior

The worst algorithm for any problem is generate-and-test. We generate complete candidate solutions and then test them for feasibility and optimality. No information from the “test” is used during candidate generation.

The best algorithm for a problem completely incorporates the “test” constraints into the solution generation so that every generated solution is feasible. Furthermore, the feasible solutions are generated in order of desirability!

How can we incorporate feasibility constraints into the operators?

Example: Efficient Generation of Pairs of Gears with Desired Ratio

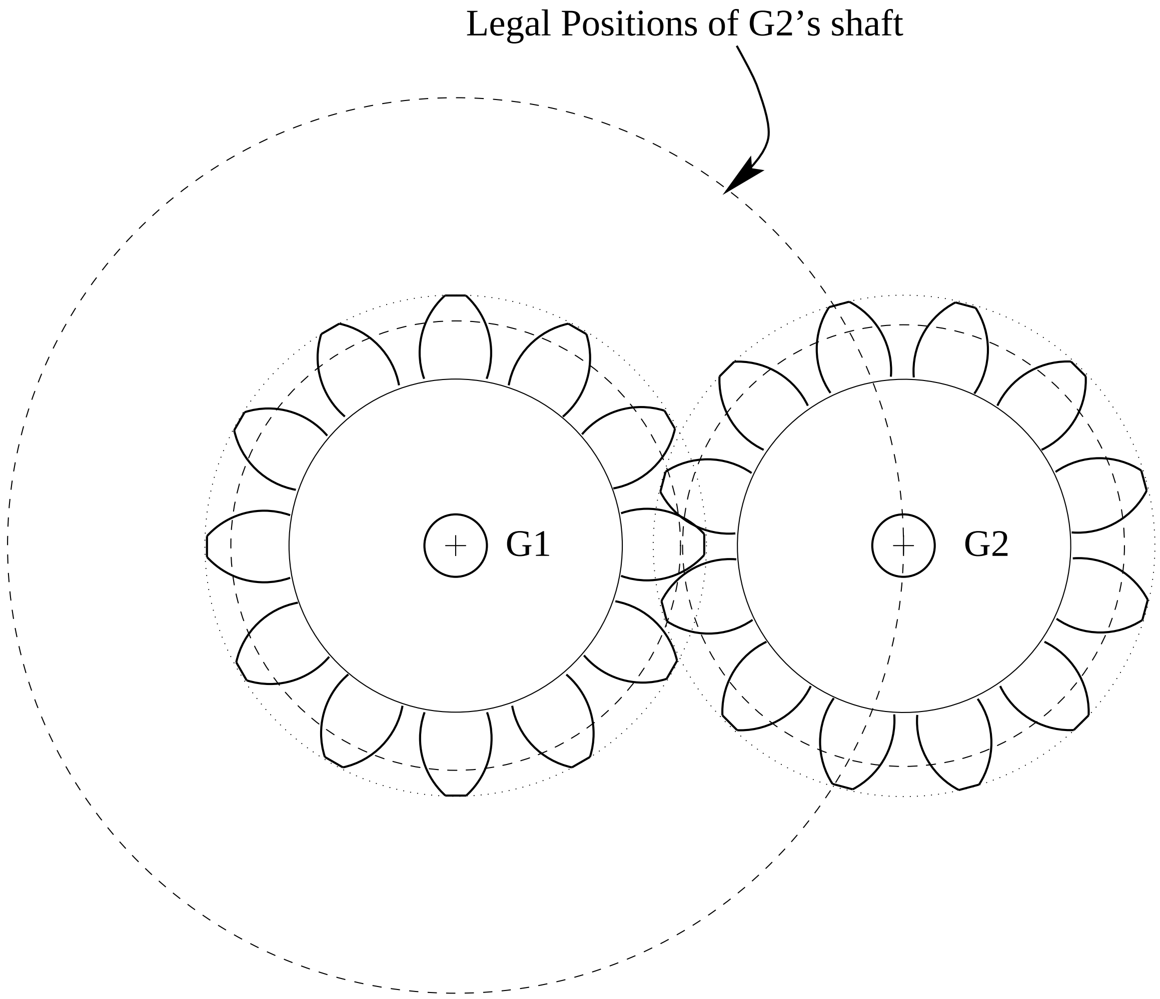
Gear 1 teeth	Gear 2 teeth
12	12
15	15
16	16
18	18
20	20
21	21
22	22
24	24
27	27
30	30

If ratio is too small, move down Gear 1 column.
If ratio is too big, move down Gear 2 column.

This takes time that is linear in the number of gears, rather than quadratic. (30 possible combinations instead of 900; 900 instead of 810,000 for four gears).

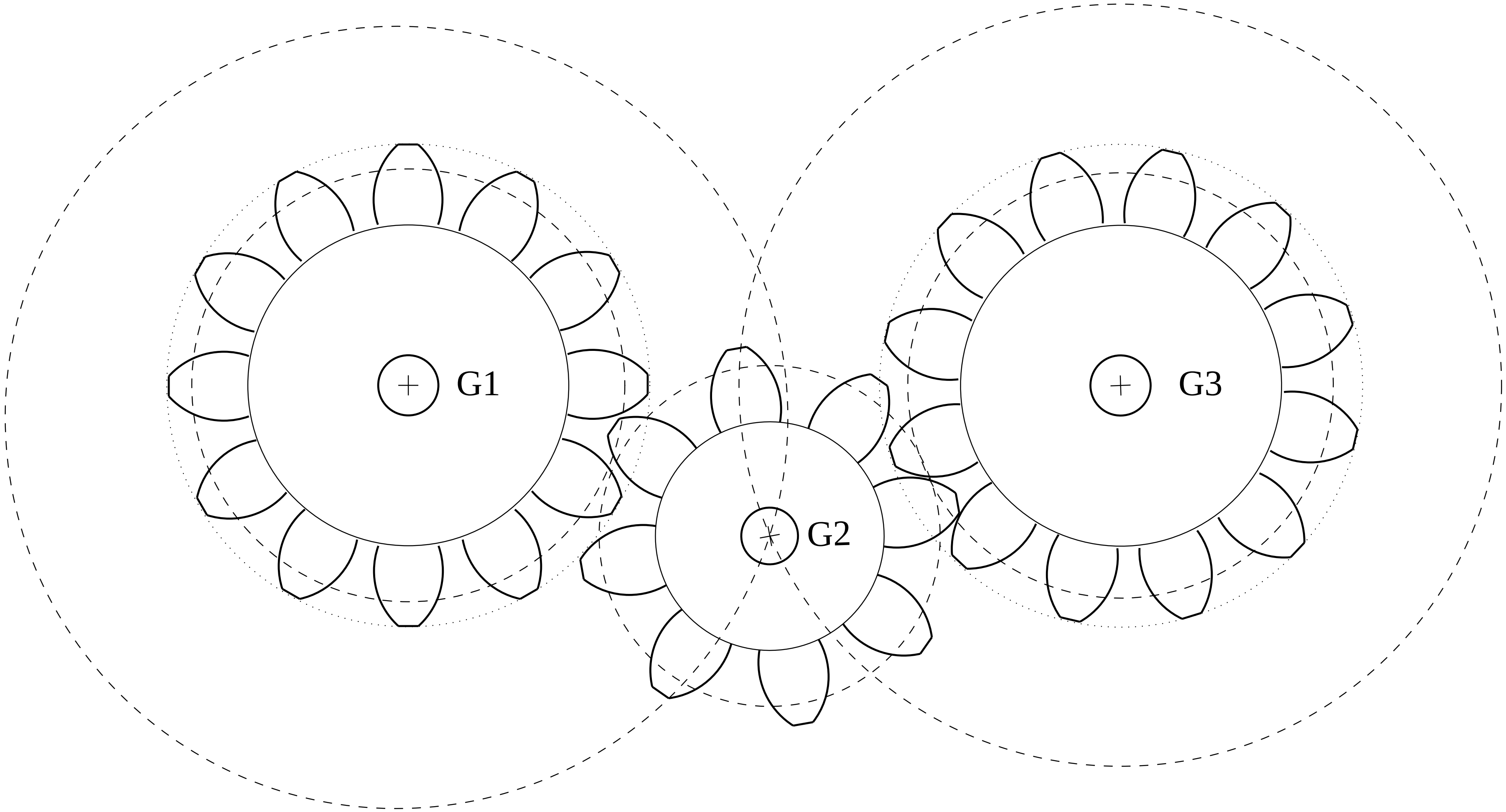
Some Constraints Can Be "Propagated"

If we have chosen gears $G1$ and $G2$ and we know the location of $G1$, then the location of gear $G2$ is constrained to lie on a circle centered at the center of $G1$ with radius equal to the sum of the pitch radii of $G1$ and $G2$.



Local Constraint Propagation

If we have chosen three gears $G1$, $G2$, and $G3$ that are to mesh and we know the locations of $G1$ and $G3$, then the location of the shaft of $G2$ is constrained to be one of two points.



Other Constraints Must Be Applied as Tests

- **Local Tests Involving *Subparts* of the Design**

  Whenever a gear is chosen and positioned, it can be tested for collision with the box and with all other gears and shafts.

- **Global Tests**

  Tests involving a global constraint must be addressed globally. We can't avoid generate-and-test behavior with global constraints.

Intermixing Generate-and-Test with Constraint Propagation (GTCP)

- **state:** A set of gears and shafts that have been selected so far plus a set of **constraints** that have been **posted** on the remainder of the design.
- **starting state:** An empty design with a set of constraints. These constraints are the constraints on the feasible solutions.
- **feasible solution:** A state with no pending constraints.
- **operators:** The operators are divided into three kinds:
  - **generation operators** Generate possible pairs, triples, quadruples, etc. of gears that satisfy current posted constraints.
  - **test operators** Test pending constraints against recently generated parts of the solution.
  - **propagation operators** Given current choices for components and some pending constraints, *solve for* correct choices for other components.

Control Structure

A form of depth-first search is usually employed. Operators are applied with the following priority.

1. **Apply any executable test operators.**
2. **Apply any executable propagation operators.** Prefer the operators that are most tightly constraining.
3. **Apply any executable generation operators.** Prefer the generators that generate over the smallest sets.

Do The Most Constraining Thing First

Generation Operators for Gear Design

- **Design Type generation operator**. Generate DesignType in the order OneOne, OneOneOne, OneTwoOne, OneOneOneOne, ...
- **Gear generation operator**. Generate the possible gears from the catalog in ascending order by size.
- **Gear tuple generation operators**. Given a number of gears k and a desired ratio R, generate all k-*tuples* of gears that give the desired ratio R. Generate in ascending order by maximum size.
- **Shaft location generation operators**. Given a set of possible locations for a shaft, generate them.

Propagation Operators for Gear Design

- **Inter-shaft distance.** Given the allowed locations of the input and output shafts, post constraints on the minimum and maximum possible distance between the shafts.
- **Input and output shaft positions.** In the OneOne design, given the input and output gears, compute the locations of the input and output shafts to satisfy the shaft location constraints. In all other designs, place the input and output shafts at their nominal positions.
- **Idler shaft positions.** Given a 3-tuple of three adjacent gears in a gear chain and the positions of the first and third shafts, compute the two possible locations of the middle shaft. Given a 4-tuple of gears where the two middle gears share an idler shaft, and given the locations of the first and third shafts, compute the two possible locations of the middle shaft.

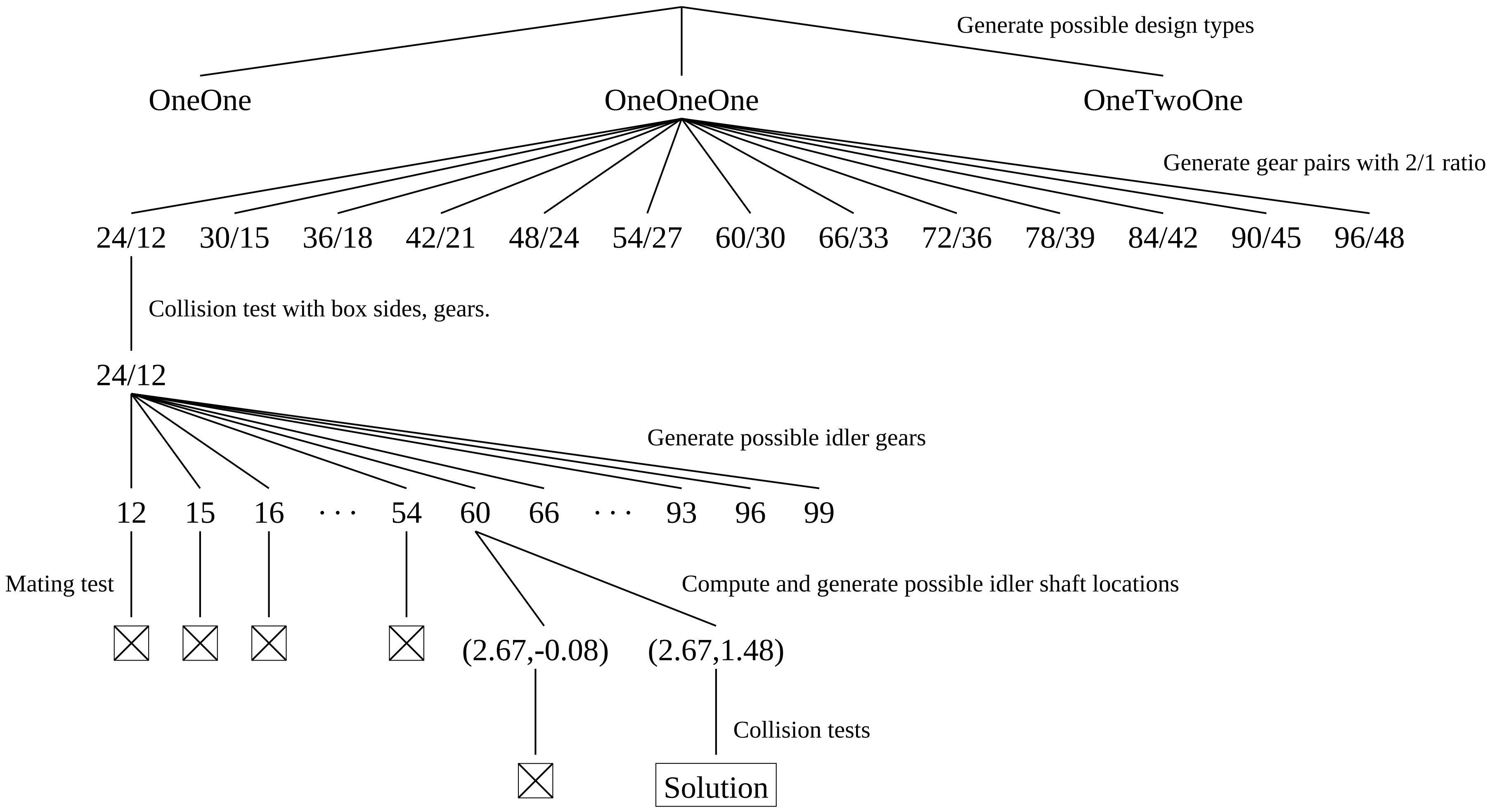
Test Operators for Gear Design

- **Gear Collision.** Given a gear and the location of its shaft, test whether it collides with the side of the box. Given a gear and the location of its shaft, and the location of any other shaft, test whether it collides with that other shaft. Given two non-mating gears, the locations of their shafts, and the positions of the gears on those shafts, test whether the gears collide.

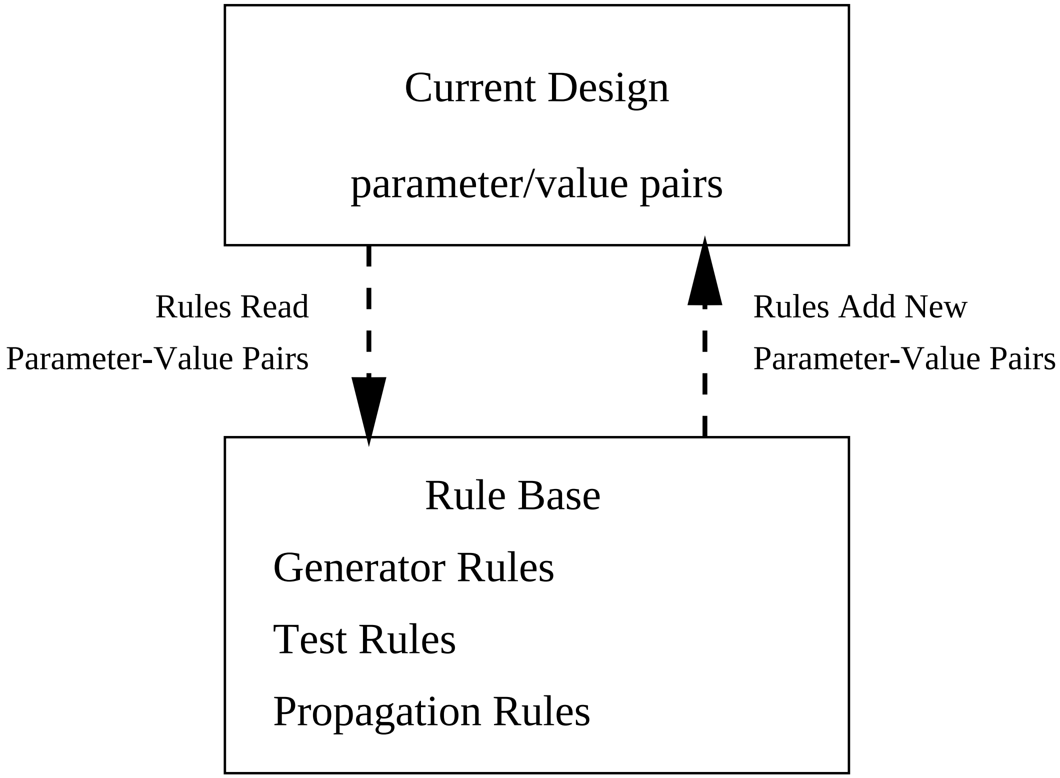
- **Gear Size.** Given the input and output gears in the OneOne design, test that the sum of their pitch radii is within the minimum and maximum input/output shaft separation constraints.

- **Idler gear size.** Given a 3-tuple of gears and the locations of the shafts of the first and third gears, test whether the middle gear is big enough to mesh with the first and third gears. (In other words, the sum of the pitch radii of the three gears must be $\geq$ the distance between the input and output shafts.)

Example Search Process



Implementing GTCP via Rule-Based Programming



Rules run whenever they can. They are sometimes called "daemons" or "consumers". This is exactly what we want for test operators and propagation operators—they should run as soon as possible during problem solving.

Parameters and Values for Gear Box Design

The entire design is defined by the values of a set of variables. These design variables are called *parameters*.

Specification Parameters	Intermediate Parameters	Design Parameters
R (desired ratio)	d (in/out shaft distance)	Design (OneOne, etc.)
boxWidth	dmin (min value of d)	inGear (input gear)
boxLength	dmax (max value of d)	outGear (output gear)
in (input shaft)		inoutPair (gear pair)
out (output shaft)		idler1 (idler shaft loc)
		idler1Gear1
		idler1Gear2
		etc.

Rule/Parameter Interaction using Object-Oriented Programming

Each rule has a *condition part* **and an** *action part*

- **Each rule is an object**
- **Each parameter is an object**
- **Assigning a value causes messages to be sent**

  When a rule assigns a new value to a parameter, that parameter notifies all rules that are "interested in" that parameter.
- **Rule firing**

  When all conditions of a rule are satisfied, then it "fires" and executes its action part, which alters other parameters.

Forward-Reasoning and Backward Reasoning

Rule-based systems can be organized as forward-reasoning systems or as backward-reasoning systems.

- **Forward Reasoning** (aka forward chaining). A rule is executed whenever its condition part is matched. If more than one rule is triggered at the same time, then test rules are preferred to propagate rules which are preferred to generator rules.
- **Backward Reasoning** (aka backward chaining). Problem solving starts with a goal. A rule is triggered if its action part matches the goal. Each condition in its condition part is then declared to be a new goal.

Implementing a Rule Language

There are two basic strategies for implementing a rule language:

- **Rule Interpreters**. A rule interpreter reads in the rules, stores them in a data structure, and executes them by traversing this data structure.
- **Rule Translators**. A translator converts the rules into another language and then compiles them. This is the strategy I used.

  The rule preprocessor reads the file `rules.r` and writes the file `rules.h`. This file is then `#included` into the main program, `gears.c`, where it is compiled.

The Rule Language

BEGINGENERATOR(<rulename>)

NEEDS(<param1 name>) NEEDS(<param2 name>) . . .

IF(<condition>)

THEN(<action>)

ENDRULE

or BEGINPROPAGATE or BEGINTEST.

The NEEDS entries set up "active values". Whenever the value of one of those parameters changes, the condition part of the rule will be evaluated to see if it is satisfied.

When the condition is satisfied, the rule is placed in the *Conflict Set*. If a condition becomes "unsatisfied", the rule is removed from the *Conflict Set*.

There are two special values: KNOWN and UNKNOWN.

Example Generator Rule

```
BEGINGENERATOR(Rule3_ioRatioLoop)
NEEDS(Design) NEEDS(inoutPair) NEEDS(R)
IF(((Design == OneOne || Design == OneOneOne)) &&
   (inoutPair == UNKNOWN))
THEN(list<catalogRatio> * candidates = catalogRatiosMatchingRatio(R());
      listIterator<catalogRatio> itr(*candidates);
      inoutPair.tryAll(itr);
      delete candidates;
      )
ENDRULE
```

This sets up a "nested loop". It will generate all possible values using the iterator itr. It generates each possible value and then executes all rules that are triggered by that new value. Those rules can signal a violation of a constraint, which causes the iterator to try the next value.

Example Propagator Rule

```
BEGINPROPAGATE(Rule5_ioPropagate)
NEEDS(inoutPair)
IF( inoutPair == KNOWN)
THEN (
    inGear = inoutPair().g1;
    outGear = inoutPair().g2;
    )
ENDRULE
```

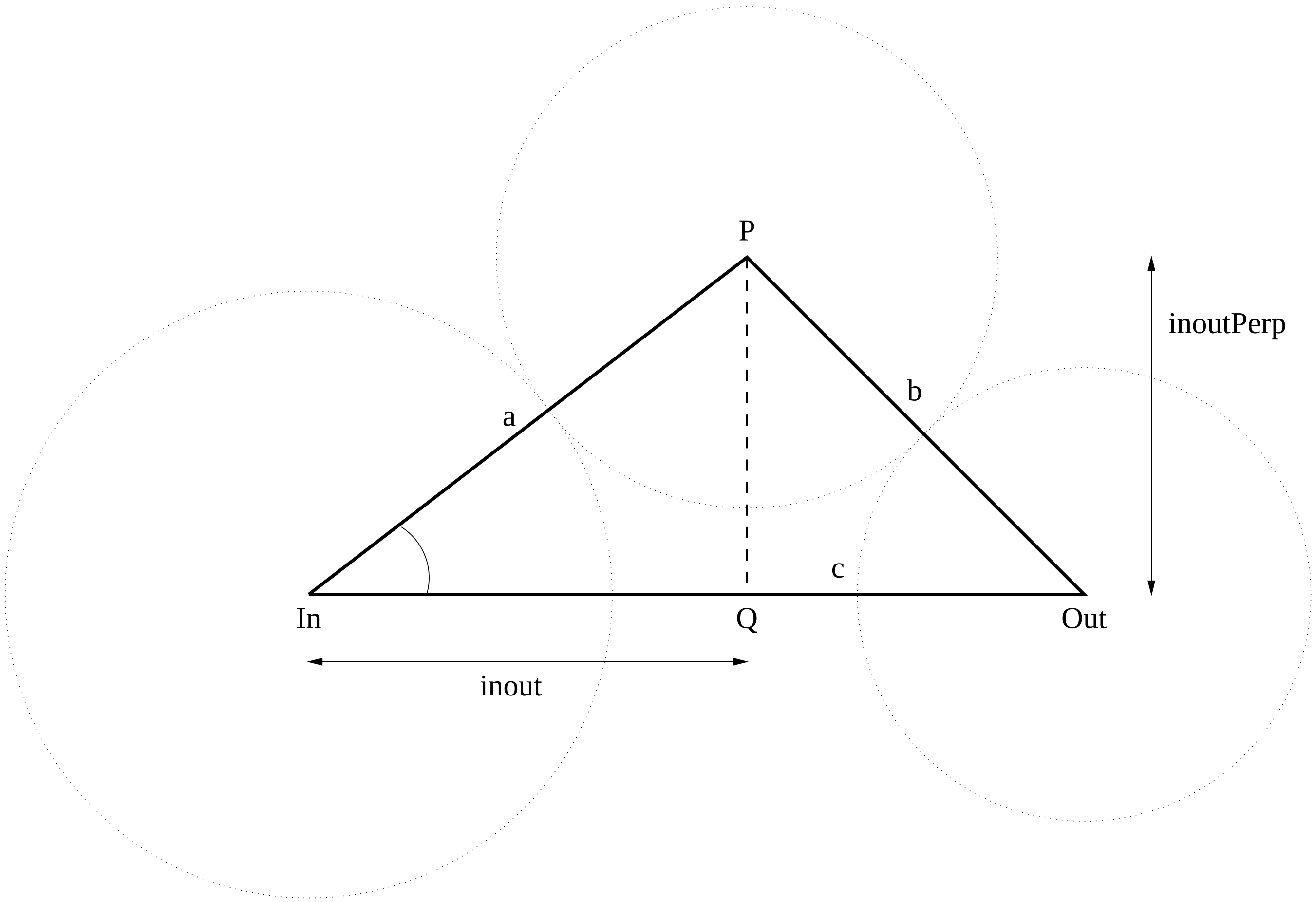
This is a trivial propagator. It extracts the two individual gears generated by the previous rule and assigns them to the inGear and outGear parameters.

Example Test Rule

```
BEGINTEST(Rule4_IOGearsCantMesh)
NEEDS(Design) NEEDS(inGear) NEEDS(outGear) NEEDS(dmin) NEEDS(dmax)
IF((Design == OneOne) &&
   (inGear == KNOWN) &&
   (outGear == KNOWN) &&
   (dmin == KNOWN))
THEN (
      test((inGear().PitchRadius + outGear().PitchRadius ≥ dmin()) &&
       (inGear().PitchRadius + outGear().PitchRadius ≤ dmax()),
       "input/output gears can't mesh");
    )
ENDRULE
```

The **test** function cause a "failure" back to the most recent choice made by a generator. This is called "back-tracking" search. The generator makes a new choice, which causes rules to be triggered again.

Combining Propagation and Generation: Computing the Idler Shaft Location



Idler Shaft Location Rule

```
BEGINGENERATOR(Rule8_GenerateIdler1)
NEEDS(Design) NEEDS(in) NEEDS(out) NEEDS(inGear) NEEDS(outGear)
NEEDS(idler1Gear1) NEEDS(idler1) NEEDS(d)
IF ((Design == OneOneOne) && (in == KNOWN) && (out == KNOWN) &&
   (inGear == KNOWN) && (outGear == KNOWN) && (idler1Gear1 == KNOWN) &&
   (idler1 == UNKNOWN) && (d == KNOWN) )
THEN ({list<point> possibleCenters;
       double r = idler1Gear1().PitchRadius, a = inGear().PitchRadius + r;
       double b = outGear().PitchRadius + r,  c = d();
       double cosIn = ((square(c) + square(a) − square(b)) / (2 ∗ a ∗ c));
       double sinIn = sqrt(1 − square(cosIn));
       double inout = a ∗ cosIn,  inoutPerp = a ∗ sinIn;
       point deltaOutIn = out() − in();
       point Q = in() + (inout / c) ∗ deltaOutIn;
       point temp = (inoutPerp / c) ∗ deltaOutIn;
       point P(−temp.y, temp.x);  // rotate by 90 degrees
       possibleCenters.add(Q + P); possibleCenters.add(Q − P);
       listIterator<point> itr(possibleCenters);
       idler1.tryAll(itr);
      }
     )
ENDRULE
```

Parameters and Parameter Types

```
ParameterType<catalogGear> Gear;
ParameterType<tolerancedPoint> Shaft;
parameter<rational> R("R");                          // input parameters
parameter<double> boxWidth("boxWidth");
parameter<double> boxLength("boxLength");
parameter<tolerancedPoint> in("input shaft", Shaft);
parameter<tolerancedPoint> out("output shaft", Shaft);
parameter<double>  dmin("dmin");                     // intermediate parameters
parameter<double>  dmax("dmax");
parameter<double>  d("d");
parameter<designType> Design("Design");              // design parameters
parameter<catalogGear> inGear("inGear", Gear);
parameter<catalogGear> outGear("outGear", Gear);
parameter<catalogRatio> inoutPair("inoutPair");
parameter<tolerancedPoint> idler1("idler1 shaft", Shaft);
parameter<catalogGear> idler1Gear1("idler1 gear 1", Gear);
parameter<catalogGear> idler1Gear2("idler1 gear 2", Gear);
```

What a Rule Looks Like

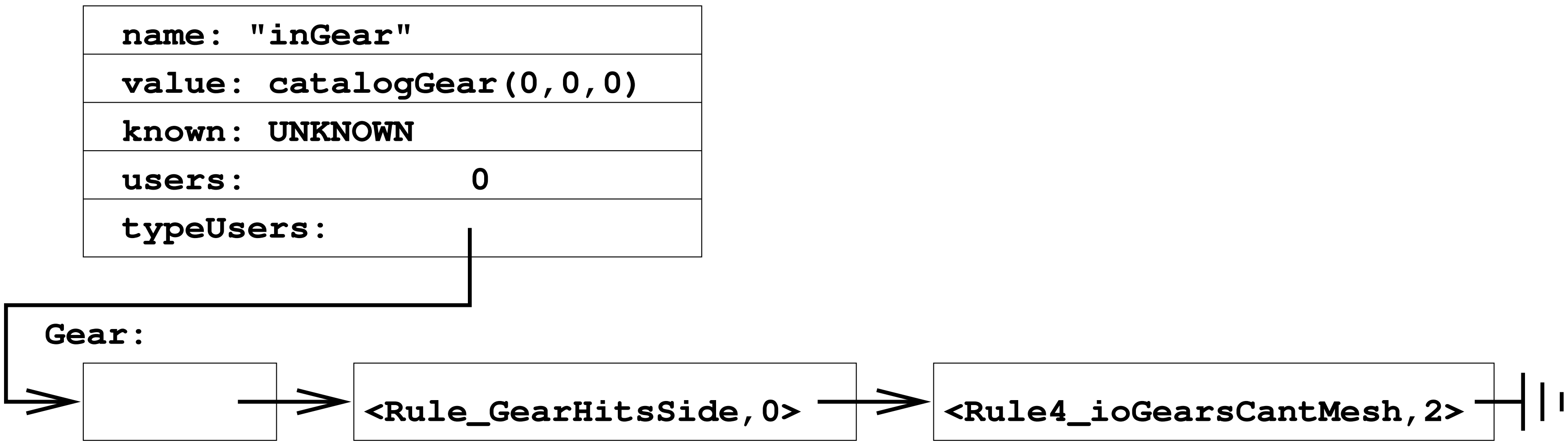
```
NEEDS(Gear)   NEEDS(boxWidth)   NEEDS(boxLength)   NEEDS(Shaft)
```

```
IF(Gear == KNOWN && boxWidth == KNOWN & boxLength == KNOWN &&
   Shaft == KNOWN && MountedOn(Gear, Shaft))

THEN
   test(
     Shaft().x + Gear().OuterRadius < boxWidth() - 0.1 ||
     Shaft().x - Gear().OuterRadius > 0.1 ||
     Shaft().y + Gear().OuterRadius < boxLength() - 0.1 ||
     Shaft().y - Gear().OuterRadius > 0.1,

     "Gear does not fit in the box")
```

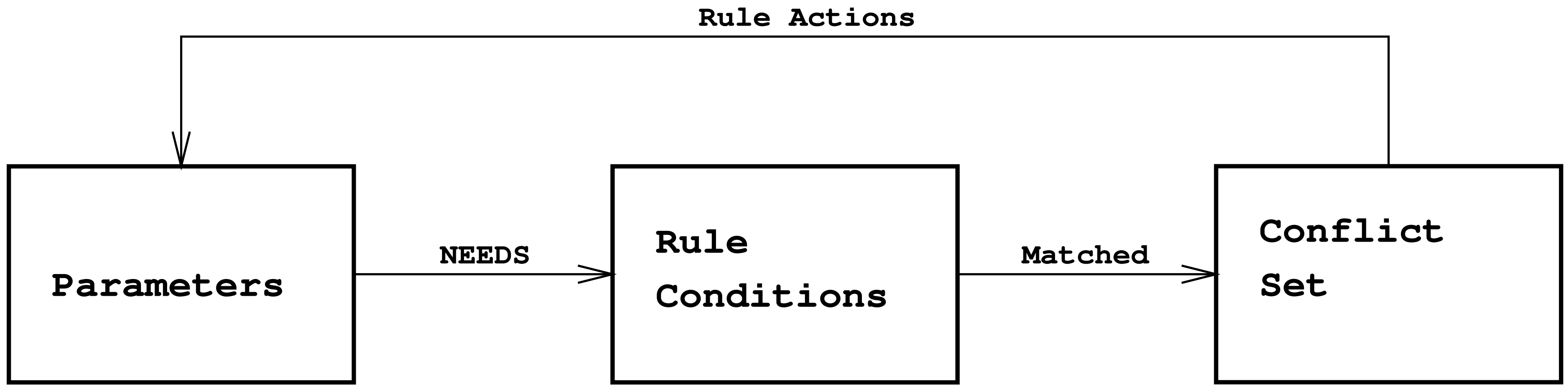
What a Parameter Looks Like

```
parameter<catalogGear> inGear("inGear", Gears);
```

inGear:



Forward Chaining Architecture



Changes in parameter values cause rule conditions to be checked.

If a rule condition matches, the rule (along with the matching parameter values) is placed on the conflict set.

After a rule finishes executing, a new rule is chosen from the conflict set, removed, and executed.

Sequence of Assignments to Active Values

inGear = UNKNOWN;

outGear = UNKNOWN;

idler1Gear1 = UNKNOWN;

idler1Gear2 = UNKNOWN;

boxWidth = UNKNOWN;

boxLength = UNKNOWN;

in = UNKNOWN;

out = UNKNOWN;

idler1 = UNKNOWN;

boxWidth = 2;

boxLength = 2;

inGear = catalogGear(24, 1.000, 1.083);

outGear = catalogGear(12, 0.500, 0.583);

in = (0.7, 0.7);

out = (1.4, 0.7);

Initial UNKNOWN Assignments

NEEDS(Gear)	NEEDS(boxWidth)	NEEDS(boxLength)	NEEDS(Shaft)
inGear:UNKNOWN outGear:UNKNOWN idler1Gear1:U idler1Gear2:U	boxWidth: UNK	boxLength: UNK	

```
IF(Gear == KNOWN && boxWidth == KNOWN && boxLength == KNOWN &&
   Shaft == KNOWN && MountedOn(Gear, Shaft))
THEN  ( test( Shaft().x + Gear().OuterRadius < boxWidth() - 0.1 &&
              Shaft().x - Gear().OuterRadius > 0.1 &&
              Shaft().y + Gear().OuterRadius < boxLength() - 0.1 &&
              Shaft().y - Gear().OuterRadius > 0.1,
              "Gear does not fit in the box"))
```

Each new value tries to form a 4-tuple with the values in the other Beta memories.

Forming Tuples

NEEDS(Gear)	NEEDS(boxWidth)	NEEDS(boxLength)	NEEDS(Shaft)
inGear:UNKNOWN outGear:UNKNOWN idler1Gear1:U idler1Gear2:U	boxWidth: UNK	boxLength: UNK	in: UNKNOWN

```
<inGear:U, boxWidth:U, boxLength:U, in:U>
<outGear:U, boxWidth:U, boxLength:U, in:U>
<idler1Gear1:U, boxWidth:U, boxLength:U, in:U>
<idler1Gear2:U, boxWidth:U, boxLength:U, in:U>
```

```
IF(Gear == KNOWN && boxWidth == KNOWN && boxLength == KNOWN &&
   Shaft == KNOWN && MountedOn(Gear, Shaft))
THEN ( test( Shaft().x + Gear().OuterRadius < boxWidth() - 0.1 &&
             Shaft().x - Gear().OuterRadius > 0.1 &&
             Shaft().y + Gear().OuterRadius < boxLength() - 0.1 &&
             Shaft().y - Gear().OuterRadius > 0.1,
             "Gear does not fit in the box"))
```

Assigning a New Value to boxWidth

- Delete the old value from the Beta Memory
- Remove any tuples from the Conflict Set involving this value in this position.
- Add the new value to the Beta Memory
- Form tuples and check the rule condition
- Add successful tuples to the Conflict Set

Assigning a New Value to boxWidth (2)

NEEDS(Gear)	NEEDS(boxWidth)	NEEDS(boxLength)	NEEDS(Shaft)
inGear:UNKNOWN outGear:UNKNOWN idler1Gear1:U idler1Gear2:U		boxLength: UNK	in: UNKNOWN

```
IF(Gear == KNOWN && boxWidth == KNOWN && boxLength == KNOWN &&
   Shaft == KNOWN && MountedOn(Gear, Shaft))
THEN ( test( Shaft().x + Gear().OuterRadius < boxWidth() - 0.1 &&
             Shaft().x - Gear().OuterRadius > 0.1 &&
             Shaft().y + Gear().OuterRadius < boxLength() - 0.1 &&
             Shaft().y - Gear().OuterRadius > 0.1,
             "Gear does not fit in the box"))
```

Assigning a New Value to boxWidth (3)

NEEDS(Gear)	NEEDS(boxWidth)	NEEDS(boxLength)	NEEDS(Shaft)
inGear:UNKNOWN outGear:UNKNOWN idler1Gear1:U idler1Gear2:U	boxWidth:2	boxLength: UNK	in: UNKNOWN

```
<inGear:U, boxWidth:2, boxLength:U, in:U>
<outGear:U, boxWidth:2, boxLength:U, in:U>
<idler1Gear1:U, boxWidth:2, boxLength:U, in:U>
<idler1Gear2:U, boxWidth:2, boxLength:U, in:U>
```

```
IF(Gear == KNOWN && boxWidth == KNOWN && boxLength == KNOWN &&
   Shaft == KNOWN && MountedOn(Gear, Shaft))
THEN ( test( Shaft().x + Gear().OuterRadius < boxWidth() - 0.1 &&
             Shaft().x - Gear().OuterRadius > 0.1 &&
             Shaft().y + Gear().OuterRadius < boxLength() - 0.1 &&
             Shaft().y - Gear().OuterRadius > 0.1,
             "Gear does not fit in the box"))
```

The First Successful Tuple

NEEDS(Gear)	NEEDS(boxWidth)	NEEDS(boxLength)	NEEDS(Shaft)
inGear:<24,...> outGear:<12,...> idler1Gear1:U idler1Gear2:U	boxWidth:2	boxLength:2	in:(.7,.7)

```
<inGear:<24,...>, boxWidth:2, boxLength:2, in:(.7,.7)>
<outGear:<12,...>, boxWidth:2, boxLength:2, in:(.7,.7)>
<idler1Gear1:UNKNOWN, boxWidth:2, boxLength:2, in:(.7,.7)>
<idler1Gear2:UNKNOWN, boxWidth:2, boxLength:2, in:(.7,.7)>
```

```
IF(Gear == KNOWN && boxWidth == KNOWN && boxLength == KNOWN &&
   Shaft == KNOWN && MountedOn(Gear, Shaft))
THEN (test( Shaft().x + Gear().OuterRadius < boxWidth() - 0.1 &&
            Shaft().x - Gear().OuterRadius > 0.1 &&
            Shaft().y + Gear().OuterRadius < boxLength() - 0.1 &&
            Shaft().y - Gear().OuterRadius > 0.1,
            "Gear does not fit in the box"))
```

```
<inGear:<24,...>, boxWidth:2, boxLength:2, in:(.7,.7)>
```

The Second Successful Tuple

NEEDS(Gear)	NEEDS(boxWidth)	NEEDS(boxLength)	NEEDS(Shaft)
inGear:<24,...> outGear:<12,...> idler1Gear1:U idler1Gear2:U	boxWidth:2	boxLength:2	in:(.7,.7) out:(1.4,.7)

```
<inGear:<24,...>, boxWidth:2, boxLength:2, out:(1.4,.7)>
<outGear:<12,...>, boxWidth:2, boxLength:2, out:(1.4,.7)>
<idler1Gear1:UNKNOWN, boxWidth:2, boxLength:2, out:(1.4,.7)>
<idler1Gear2:UNKNOWN, boxWidth:2, boxLength:2, out:(1.4,.7)>
```

```
IF(Gear == KNOWN && boxWidth == KNOWN && boxLength == KNOWN &&
   Shaft == KNOWN && MountedOn(Gear, Shaft))
THEN ( test( Shaft().x + Gear().OuterRadius < boxWidth() - 0.1 &&
             Shaft().x - Gear().OuterRadius > 0.1 &&
             Shaft().y + Gear().OuterRadius < boxLength() - 0.1 &&
             Shaft().y - Gear().OuterRadius > 0.1,
             "Gear does not fit in the box"))
```

```
<inGear:<24,...>, boxWidth:2, boxLength:2, in:(.7,.7)>

<outGear:<12,...>, boxWidth:2, boxLength:2, out:(1.4,.7)>
```

The parameter class

```
template <class T>
class parameter: public parameterBase {
public:
  T value;
  int operator == ( T val);
  int operator == ( parameter<T> & pt);
  int operator == ( specialParameterValue & val);
  void operator = ( T newvalue);
  virtual void operator = ( specialParameterValue & val);
  T & operator()();
  void tryAll(iterator<T> & itr);
};
```

The ParameterBase class

```
class parameterBase
{
public:
  parameterBase(string);
  parameterBase(string, ParameterTypeBase &);

  void addUser(consumer * con, int argumentPosition);
  void notifyConsumers();

protected:
  string name;
  specialParameterValue known;
  ParameterTypeBase  users;      // linked list of consumer*
  ParameterTypeBase * typeUsers; // linked list of consumer*
};
```

How Active Values Work

```
template <class T>
void parameter<T>::operator = (T newvalue)
{
  known = KNOWN;
  value = newvalue;
  if (TRACE) {
    cout << "Setting parameter " << name << " to be " << value <<
      endl;
  }
  notifyConsumers();
}
```

Notifying Consumers

```
void parameterBase::notifyConsumers()
{
  listIterator<ConsumerArgumentPair> itr(users);
  // retract any existing values of this parameter from beta memories
  for (itr.init(); !itr; ++itr) itr().first->retract(this, itr().second);
  if (typeUsers) {
    listIterator<ConsumerArgumentPair> pitr(*typeUsers);
    for (pitr.init(); !pitr; ++pitr) pitr().first->retract(this, pitr().second);
  }
  consumer::retractFromConflictSet(this);
  for (itr.init(); !itr; ++itr) itr().first->alert(this, itr().second);
  if (typeUsers) {
    listIterator<ConsumerArgumentPair> pitr(*typeUsers);
    for (pitr.init(); !pitr; ++pitr) pitr().first->alert(this, pitr().second);
  }
}
```

Consumers

```
class consumer {
public:
  consumer(): alertBusy(0), BetaMemory(0), BetaItr(0), NArgs(0) {};
  virtual void alert(parameterBase *, int position);
  virtual void retract(parameterBase *, int position);
  virtual int condition(vector<parameterBase *> args) = 0;
  virtual void action(vector<parameterBase *> args) = 0;
  static void fireConflictSet();
  static void retractFromConflictSet(parameterBase * p);
protected:
  vector<parameterList>  BetaMemory;
  vector<parameterIterator *> BetaItr;
  int endOfIteration;
  void BetaInit();
  void init();
  int  ok(int ignoredPosition);
  int step(int ignoredPosition, int index = 0);
  static list<activationRecord> conflictSet;
};
```

Alerting a Consumer

```
void consumer::alert(parameterBase * p, int position)
{
  BetaMemory[position].add(p);
  for (init(); ok(position); step(position)) {
    argumentVector * args = new argumentVector(NArgs);
    for (int i = 0; i < NArgs; i++) (*args)[i] = (*(BetaItr[i]))();
    if (condition(*args)) conflictSet.add(activationRecord(this, args));
  }
}
```

Executing the Generator

```
template <class T> void parameter<T>::tryAll(iterator<T> & itr)
{
  for(itr.init(); !itr; ++itr) {
    (*this) = itr();
    try { fireConsumers(); }
    catch (Failure & u) { }
    // undo all assignments on the stack down to and including us
    while (!parameterStack.isEmpty()) {
      parameterBase * next = parameterStack.pop();
      // make this variable unknown, which will force retraction of
      // any previous value.
      (*next) = UNKNOWN;
      if (next == this) break;      // stop when we reach ourselves.
    }
  }
  throw Failure();
}
```

The test function

```
class Failure
{
public:
  Failure() {};
};

void test(int condition, char * message)
{
  if (!condition) {
    if (TRACE) cout << message << endl;
    throw Failure();
  }
}
```

The Main Program

```
int main ()
{
  // read in the gear catalog, initialize gear pairs...
  // read in the problem description
  cout << "Enter box width: " ;
  cin >> boxWidth;               // etc...
  consumer::fireConflictSet();
  // print the solution
  if((Design == OneOne) && (inGear == KNOWN) && (outGear == KNOWN)) {
    cout << endl << "Solution:" << endl;
    cout << "Input Gear: " << inGear << endl;
    cout << "Output Gear: " << outGear << endl;
  }
  else if // ... other Design types
  else cout << endl << "Unable to find solution" << endl;
  return 0;
}
```

Backward Reasoning Systems

Starting Goal: LegalDesign()

```
IF (Design = OneOne && Gear(inGear) && Gear(outGear) &&
    ratioOK(inGear, outGear) &&
    sizeOK(inGear, outGear) &&
    shaftsOK(inGear, outGear, inSpecification, outSpecification, in, out) &&
    noCollisions(inGear, in, boxWidth, boxLength) &&
    noCollisions(outGear, out, boxWidth, boxLength))
THEN LegalDesign()

IF G = catalogGear(12, 0.500, 0.583)
THEN Gear(G)

IF G = catalogGear(15, 0.625, 0.708)
THEN Gear(G)  // etc.

IF (rational(G1.NumberOfTeeth, G2.NumberOfTeeth) == R)
THEN ratioOK(G1, G2);
```

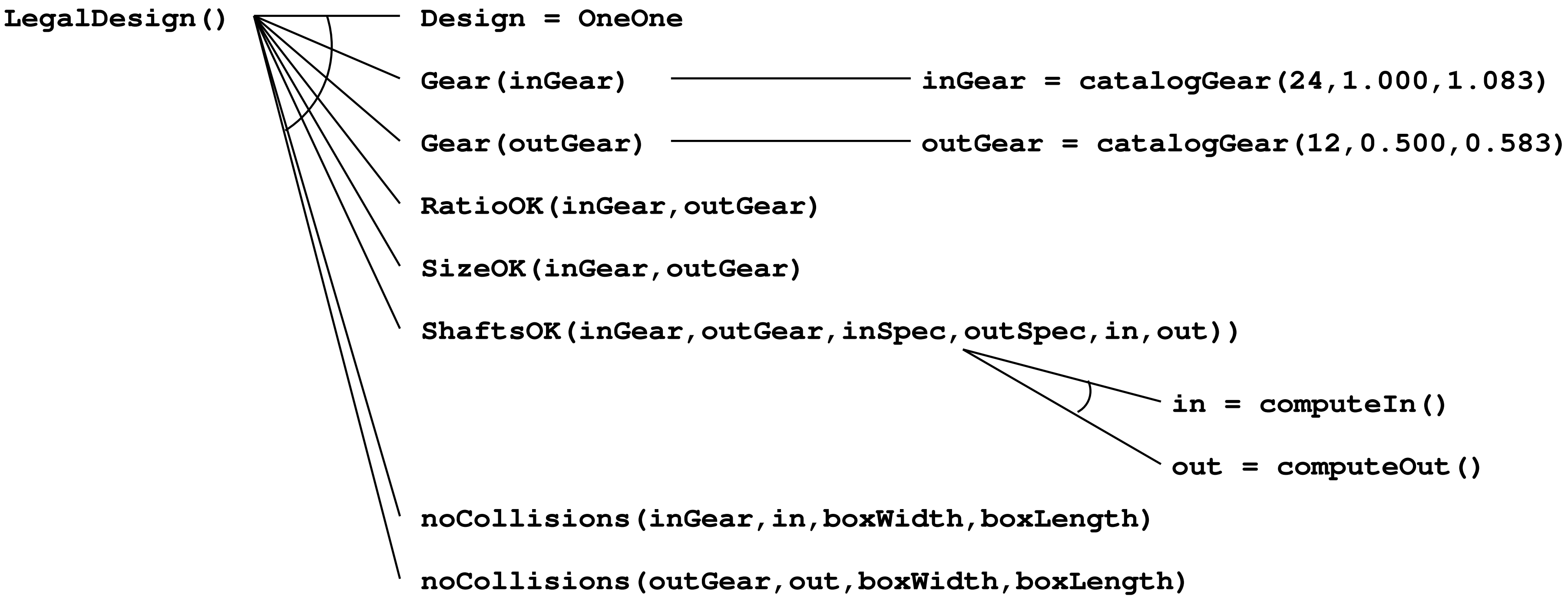
Backward Reasoning Systems (continued)

IF ((G1.PitchRadius $+$ G2.PitchRadius $\geq$ dmin) &&
    (G1.PitchRadius $+$ G2.PitchRadius $\leq$ dmax))
THEN sizeOK(G1, G2)

IF ((S.x $+$ G.OuterRadius $+$ 0.1 $<$ width) &&
    (S.x $-$ G.OuterRadius $-$ 0.1 $>$ 0) &&
    (S.y $+$ G.OuterRadius $+$ 0.1 $<$ length) &&
    (S.y $-$ G.OuterRadius $-$ 0.1 $>$ 0))
THEN noCollisions(G, S, width, length)

IF ((in $=$ computeIN(inGear.PitchRadius, outGear,PitchRadius, inSpec, outSpec)) &&
    (out $=$ computeOut(inGear.PitchRadius, outGear,PitchRadius, inSpec, outSpec)))
THEN ShaftsOK(inGear, outGear, inSpec, outSpec, in, out)

Backward Chaining Search



Backward Chaining and Design

Notice that in backward chaining, we must explicitly invoke the test operators. We must be able to anticipate when those operators could be applied. In forward chaining, the operators will automatically apply whenever they can. The same is true for propagation operators.

Another disadvantage of backward chaining is that all of the relevant design parameters must be passed as arguments to the subgoals. For large, complex designs, this is awkward.

An advantage of backward chaining is that the flow of control is easier to understand. Each subgoal is very much like a subroutine call. The Prolog language actually adopts this as its function call mechanism.

Another Example of Backward Chaining: Makefiles

A simple backward chaining rule system is the Makefile system. A Makefile consists of a series of entries of the form:

$\langle target \rangle$: $\langle subgoal1 \rangle$ $\langle subgoal2 \rangle$ ...
    $\langle action1 \rangle$
    $\langle action2 \rangle$

For example, the Makefile for this assignment contains the line

```
gears: gnuplot.o gears.o consumer.o parameter.o ../lib/clstring.o \
       point.o catalog.o ../lib/rational.o special.o predicate.o test.o
        $(CC) $(CFLAGS) special.o gnuplot.o consumer.o parameter.o \
           catalog.o point.o ../lib/clstring.o ../lib/rational.o \
           predicate.o gears.o test.o -lm -lc -o gears
```

This says that in order to make the main program, `gears`, you need to already have the files `gnuplot.o`, `gears.o`, and so on. Once you have those files, you can construct `gears` by issuing the indicated command.

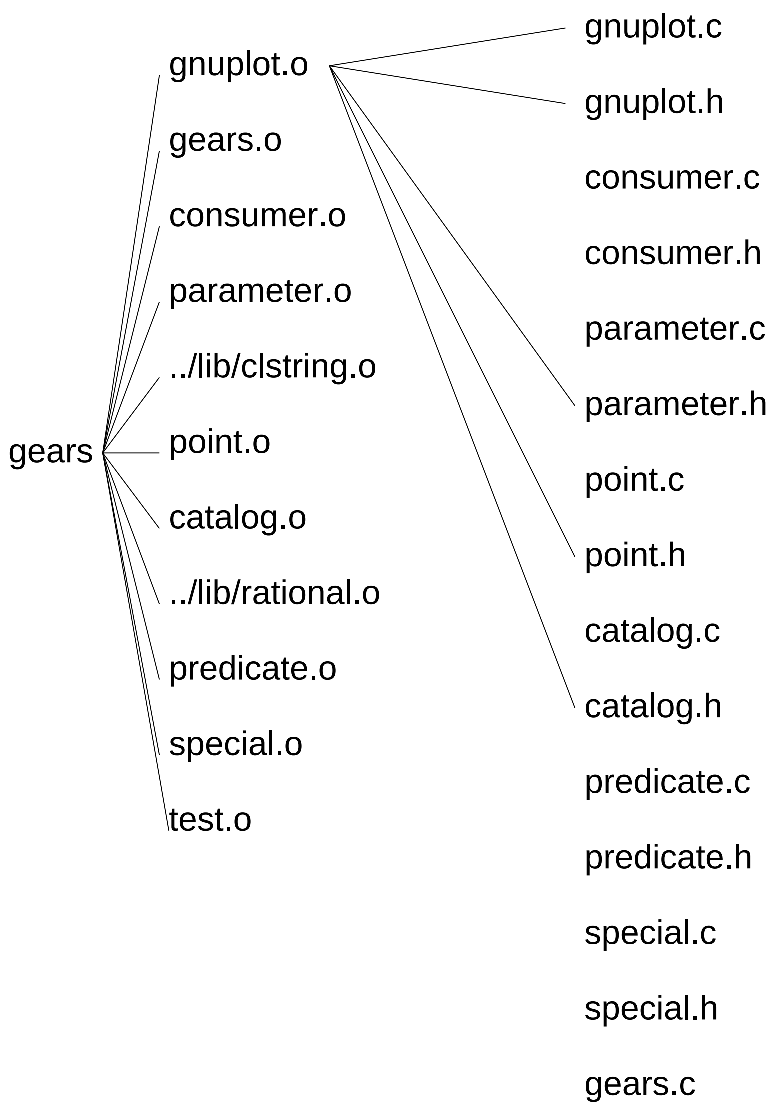
Backward Chaining

Suppose that we had installed the **`gears`** program once and that now we edit the file **`rules.r`**. What will happen when we issue the command **`make gears`**?

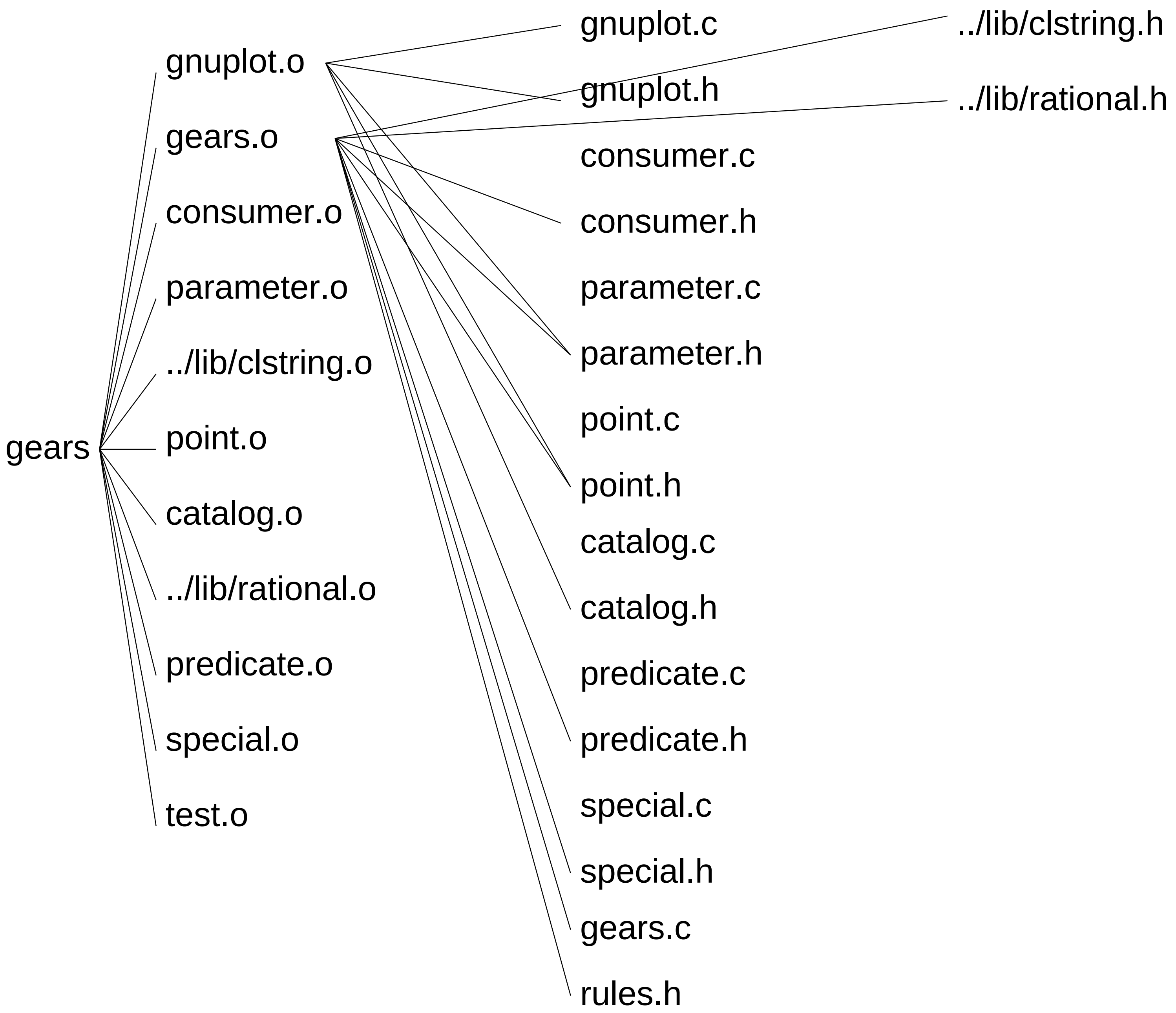
`make` works according to the following code:

```
date make(string programName)
{
  listIterator<string> pre(prequisites(programName));
  date mostRecentChange = TheYearZero;
  for(pre.init(); !pre; ++pre) {
    mostRecentChange = max(make(pre()), mostRecentChange);
    }
  if (!exists(programName) ||
     mostRecentChange > date(programName)) {
    executeActions(programName);
    mostRecentChange = date(programName);
  }
  return mostRecentChange;
}
```

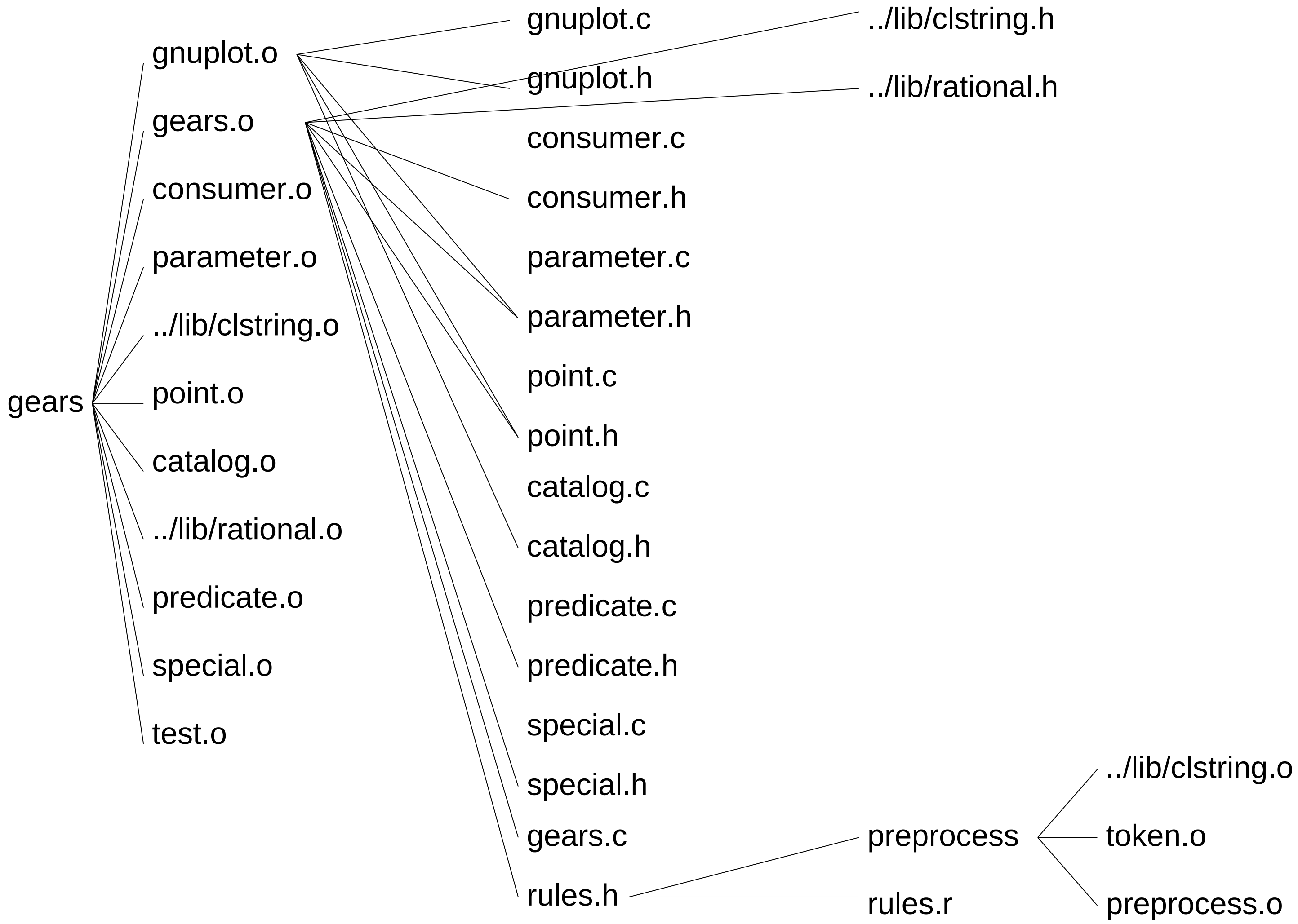
Backward Chaining (2)



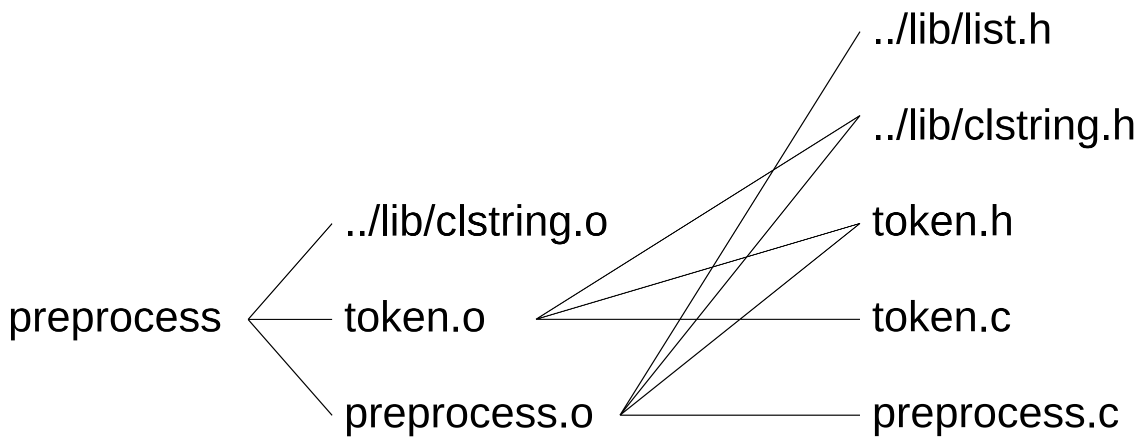
Backward Chaining (3)



Backward Chaining (4)



Backward Chaining (5)



Remembered Computations

`make` maintains a memory of what subgoals it has already processed. Before backward chaining, it checks to see if it has already processed this program once before. If so, it just returns the stored date.

```
vector<pair<string, date> > memory;
date make(string programName)
{
  date olddate;
  if(lookup(memory, programName, olddate)) return olddate;
  listIterator<string> pre(prequisites(programName));
  date mostRecentChange = TheYearZero;
  for(pre.init(); !pre; ++pre) mostRecentChange = max(make(pre()), mostRecentChange);
  if (!exists(programName) || mostRecentChange > fileDate(programName)) {
    executeActions(programName);
    mostRecentChange = fileDate(programName);
    remember(programName, mostRecentChange);
  }
  return mostRecentChange;
}
```

Backward Chaining: Execution

What you see on your screen is:

```
% make gears
rm -fr rules.h
preprocess <rules.r >rules.h
g++ -g -c gears.c
g++ -g special.o gnuplot.o consumer.o parameter.o catalog.o \
    point.o ../lib/clstring.o ../lib/rational.o predicate.o
    test.o gears.o -lm -lc -o gears
```

The first two lines are the action for `rules.h`. The next line is the action for `gears.o`, and the last line is the action for `gears`.

Advantages for Rule-Based Programming

- **Modularity**. Each rule (if properly written) tests its entire context and can be read independently of any other rule. Rules can be added or deleted incrementally.
- **Uniformity**. All rules have the same structure. This makes it easy to develop tools for entering, editing, and debugging rules.
- **Naturalness**. Many researchers in cognitive psychology have found rules to be a very good representation of human expertise. According to their theories, an expert is someone who has approx. 50,000 rules that describe what to do in particular situations.

Disadvantages of Rule-Based Programming

There are many aspects of programs that must be expressed for the computer:

- **Domain Structure**. Representing how various parts of the problem are related to one another. E.g., how the gears are mounted on the shafts.
- **Data Flow**. How information about one data object flows to another.
- **Control Flow**. The order in which things happen. How one function calls another.

Rules describe local data flow. Control flow is determined dynamically as parameters are given values. The domain structure is nearly invisible, although various methods have been developed that couple rules with object-oriented programming.

The biggest problem with rules is therefore unanticipated control flow. This is also an advantage.

Design Summary

- **Design is Difficult**. Many different kinds of constraints. Must deal with tradeoffs. Infinite and continuous state spaces.
- **Routine Configuration Design**. Select and arrange existing components.
- **Avoiding Generate-and-Test Behavior**.
  - **Incorporate constraints into generators**
  - **Propagate constraints from one variable to another**
  - **Apply tests early**
- **Rule-Based Programming**. Propagates constraints and applies tests as soon as possible. Constraints and tests can be stated declaratively.